

2011-01-01

# MVC-based content management on the cloud

Drusinsky, Doron

Monterey, California. Naval Postgraduate School

NPS-CS-11-001


# NAVAL POSTGRADUATE SCHOOL

## MONTEREY, CALIFORNIA

**MVC-based Content Management on the Cloud**

by

Doron Drusinsky

January 2011

**Approved for public release; distribution is unlimited**

Prepared for: Office of the DoD Chief Information Officer
1851 S. Bell St., Suite 600
Arlington, VA 22202

THIS PAGE INTENTIONALLY LEFT BLANK

**NAVAL POSTGRADUATE SCHOOL**
**Monterey, California 93943-5000**

Daniel T. Oliver
President

Leonard A. Ferrari
Executive Vice President and
Provost

This report was prepared for and funded by the Office of the Department of Defense Chief Information Officer.



This report was prepared by:

_____
Doron Drusinsky
Associate Professor of Computer Science

Reviewed by:

_____
Peter J. Denning, Chairman
Department of Computer Science

Released by:

_____
Karl A. van Bibber
Vice President and Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| **1. REPORT DATE** | **2. REPORT TYPE** | **3. DATES COVERED** *(From - To)* |
|---|---|---|
| January 2011 | Technical Report | Oct 1 – Nov 30, 2010 |

**4. TITLE AND SUBTITLE**
MVC-based Content Management on **the Cloud**

**5a. CONTRACT NUMBER**
DWAM00390

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**

**6. AUTHOR(S)**
Doron Drusinsky

**5d. PROJECT NUMBER**

**5e. TASK NUMBER**

**5f. WORK UNIT NUMBER**

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
The Naval Postgraduate School
1411 Cunningham Road, Monterey, CA 93943

**8. PERFORMING ORGANIZATION REPORT NUMBER**
NPS-CS-11-001

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
**Office of the Department of Defense**
Chief Information Officer
1851 S. Bell Street, Suite 600
Arlington, VA 22202

**10. SPONSOR/MONITOR'S ACRONYM(S)**
DoD CIO

**11. SPONSOR/MONITOR'S REPORT NUMBER(S)**

**12. DISTRIBUTION / AVAILABILITY STATEMENT**
Approved for public release; distribution unlimited.

**13. SUPPLEMENTARY NOTES**
The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

**14. ABSTRACT**
Cloud computing describes a new distributed computing paradigm for IT data and services that involves over-the-Internet provision of dynamically scalable and often virtualized resources. While cost reduction and flexibility in storage, services, and maintenance are important considerations when deciding on whether or how to migrate data and applications to the cloud, large organizations like the Department of Defense need to consider the organization and structure of data on the cloud and the operations on such data in order to reap the full benefit of cloud computing. This report describes a cloud adaptation of Model View Controller (MVC) software engineering architectural pattern and its effect on content management in the cloud. We propose an architecture that separates the model, view, and controller aspects of a document thereby allowing greater flexibility, portability, and interoperability for document objects.

**15. SUBJECT TERMS**
Cloud computing, model view controller, architecture, object and content management, document sharing

| **16. SECURITY CLASSIFICATION OF:** | | | **17. LIMITATION OF ABSTRACT** | **18. NUMBER OF PAGES** | **19a. NAME OF RESPONSIBLE PERSON** |
|---|---|---|---|---|---|
| | | | | | Doron Drusinsky |
| **a. REPORT** | **b. ABSTRACT** | **c. THIS PAGE** | UL | 19 | **19b. TELEPHONE NUMBER** *(include area code)* |
| Unclassified | Unclassified | Unclassified | | | 831-656-2168 |

**Standard Form 298 (Rev. 8-98)**
**Prescribed by ANSI Std. Z39.18**

THIS PAGE INTENTIONALLY LEFT BLANK

# ABSTRACT

Cloud computing describes a new distributed computing paradigm for IT data and services that involves over-the-Internet provision of dynamically scalable and often virtualized resources. While cost reduction and flexibility in storage, services, and maintenance are important considerations when deciding on whether or how to migrate data and applications to the cloud, large organizations like the Department of Defense need to consider the organization and structure of data on the cloud and the operations on such data in order to reap the full benefit of cloud computing. This report describes a cloud adaptation of the Model View Controller (MVC) software engineering architectural pattern and its effect on content management in the cloud. We propose an architecture that separates the model, view, and controller aspects of a document thereby allowing greater flexibility, portability, and interoperability for document objects.

# 1. Introduction

Model–View–Controller (MVC) is a software architecture and an architectural pattern used in software engineering. The pattern isolates *domain logic* (the logic of a software application) from the *user interface* (input and presentation), permitting independent development, testing and maintenance of each (separation of concerns).

The *model* is used to manage information and notify observers when that information changes. The model is the domain-specific representation of the data upon which the application operates. Domain logic adds meaning to raw data (for example, calculating whether today is the user's birthday, or the totals, taxes, and shipping charges for shopping cart items). When a model changes its state, it notifies its associated views so they can be refreshed.

The *view* renders the model into a form suitable for interaction, typically a user interface element. Multiple views can exist for a single model for different purposes. A viewport typically has a one to one correspondence with a display device and knows how to render to it.

The *controller* receives input and initiates a response by making calls on model objects. A controller accepts input from the user and instructs the model and viewport to perform actions based on that input.[1]

Figure 1 depicts the workflow sequence diagram using MVC for web-based applications[2]. Its workflow begins with an *http* request to the controller; the controller then assembles a composite view that consists of model data and formatting data and sends it back to the browser via http for end user presentation. Note the following:

- The Web browser is not the view; rather, it is but a *canvas* on which the view is rendered, similar to the monitor not being the view, but just a display device. This explains the reason the entry point to the MVC triad in the sequence diagram is the controller and not the view, the view being just a painting (albeit dynamic) on a canvas.
- Although Fig. 1 refers to a database as the storage medium for the model, the model can be implemented using raw files, such as structured XML files. Likewise, the DML class box (Data Manipulation Language within SQL) is specific to the choice of a database in Fig. 1.
- The controller is the entry point to the MVC triad. This may be a confusing point, because often, readers expect the view to be the entry point. The view however, has no intelligence; all intelligence is embedded in the controller.

[1] http://en.wikipedia.org/wiki/Model%E2%80%93View%E2%80%93Controller

[2] http://www.tonymarston.net/php-mysql/model-view-controller.html

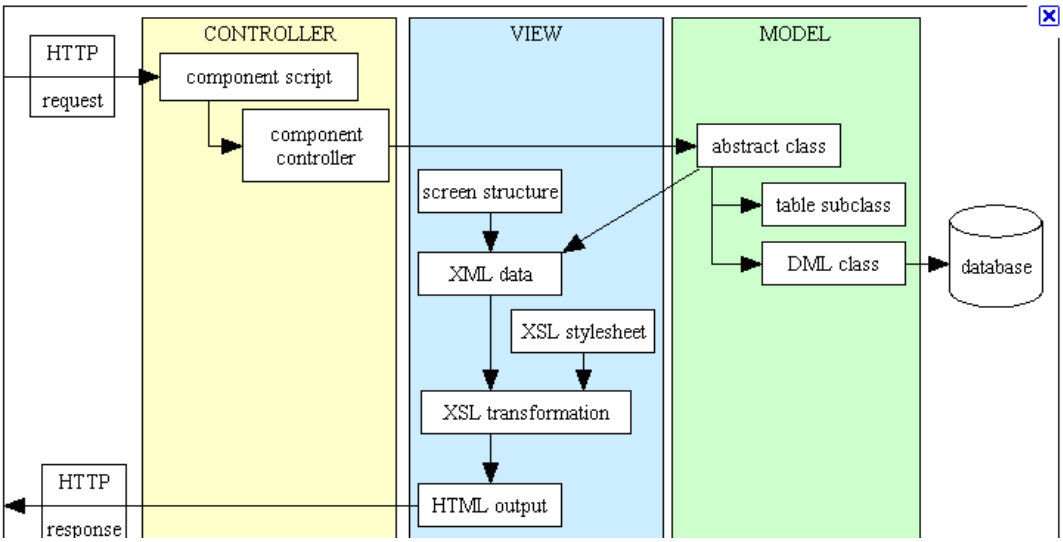


Figure 1. Sequence diagram for MVC based workflow for a web application

This report describes the benefit of applying MVC principles to content management in the cloud. As with traditional software engineering, MVC provides the following benefits when applied to document objects: flexible, late-binding of a view to the data, improved separation of concerns for storage and query purposes (e.g. storing data separate from views) With the advent of the anticipated transition to the cloud, we believe it to be the right time to consider such an approach.

It is important to note that our suggested MVC approach is not a mere recycling of the older component document approach, in which content components are dynamically assembled for different business requirements. While the two approaches share the promise of catering for dynamic creation of documents, the MVC approach contains more advantages as well as a proven track record in the software engineering world.

## 2. MVC for Documents

Consider a typical MS Word, MS Excel, MS Power Point, or Adobe PDF document. It packages all three MVC aspects namely, the *model*, the *view*, and the *controller*, in a single file. The model part is the raw data, the view part is rendering information such as color, positioning and font size, and the controller part is business logic, such as overhead rates or county tax rate formulae within a spreadsheet.

Recent changes to some file formats (most noticeably the Office Open XML format for recent versions of Microsoft Office such as .docx, .pptx, and .xlsx) use an open standard thereby making those files accessible through a plurality of applications; – a step that is seemingly in concert with the MVC pattern. Office Open XML files are actually zipped directories with various information aspects recorded in separate files within that directory. Nevertheless, even with such recent document file formats, most of

the benefits of MVC are missed out, as discussed below. The structure of Office Open XML and its drawbacks as it pertains to MVC are discussed in section 3.

We first introduce the MVC controller aspect to the document world. Suppose Alice has data for an expense report (ER), while Bob has data for a purchase order (PO). Clearly, state and county sales taxes are used in both documents. The prevailing approach is for those tax rates to be embedded in each document. This is clearly a rigid, brute force approach that requires changing each document whenever state or county tax rates change; it is also an obvious duplication of effort. In addition, if Alice and Bob want to create reusable templates for their respective documents, each end user will need to customize those tax rates according to their geographical location, requiring further manual changes to each document. It is easy to see how such ER's and PO's become obsolete after a short period of time, especially in a large organization with frequently changing business rules. We normally view the existence of such obsolete documents as *something that goes with the territory*, so to speak, and we often keep them for auditing purposes.

In contrast, an MVC approach to this situation would separate the tax related business logic from the data. Consequently, neither PO template nor ER template will contain any verbatim tax rate. Rather, they would refer to a business logic application (denoted as *TaxRateApp*) that automatically calculates the tax rates for the respective end-user depending on her location, time, and other relevant parameters. Note that tax rate can be considered as data, but for *TaxRateApp, not for the PO or ER templates* Note that business logic is calculated using a context-sensitive approach, namely in the context of a parameterized time and location; hence, when the end-user's county changes its sales tax rate, the end user will automatically benefit from templates that use an updated tax rate - because *TaxRateApp* automatically grabs the latest and greatest tax rate, without Bob and Alice needing to make any change to their respective templates. In addition, the end user is still able to render their documents in an auditing mode, i.e., based on older tax rates, by providing *TaxRateApp* with the appropriate time and location parameters.

The *view* aspect in the MVC approach is about separating the presentation of the enhanced data, namely data (model) enhanced with business rules (controller), from the model and controller. While it is possible to render contemporary documents using more than one application (e.g., Word documents in Acrobat, Google Docs, or Open Office), MVC offers greater flexibility, as follows. With the prevailing situation, all rendering information is part of the document (whether in a single file or a zipped directory), resulting in almost identical representation of the document whether opened by one application or by the other. In contrast, contemporary users (and other data readers, e.g. Business Intelligence document aggregators) need viewing capability that uses the intelligence and business logic embedded in the controller. For example, consider three instances of Alice's ER template, in the U.S, Japan, and India. The Japanese viewer will highlight travel dates that overlap Japanese holidays in red, i.e., the viewer is informed by the controller as to which dates represent holidays. Similarly, the Indian viewer will highlight travel dates that overlap an Indian holiday in some color other than red, red symbolizing purity in India, much like white does in western cultures. It is important to note that while modern applications use localization to potentially achieve a similar local-

based flexibility, they really do so for two types of properties only, namely location (using localization), and time (using localized calendars). They are devoid of an ability to integrate custom business logic based on other parameters such as the size of the organization (e.g., a government policy aimed at small business), locality (e.g., cost of living local), gender, and age groups (e.g., age based lingo).

The benefits of applying the MVC pattern to cloud content are:

1. Improved robustness. Having externalized (outside the document) business logic, means that fewer documents become stale, obsolete, and incorrect as they move within geographical and temporal spaces.

2. Improved flexibility. Custom business logic enables automated document-level integration of flexible concerns such as organization locals, weather, age groups, gender, election year information, etc.

3. Improved collaboration. Externalized business logic enables collaborative group-level decisions.

## 3. Microsoft Office Open XML (OOXML)

Every OOXML file is a ZIP archive containing many other files. Office-specific data is stored in multiple XML files inside that archive. This is in direct contrast with the old WordML and SpreadsheetML formats which were single, non-compressed XML files.

The Office Open XML specification has been standardized by Ecma in 2006.[3] A later edition was standardized in 2008 by ISO and IEC as an International Standard (ISO/IEC 29500); this edition is still not implemented in any products.

In Microsoft's terminology, an OOXML ZIP file is called a *package*. Files inside that package are called *parts*. Every part has a defined content type and there are no default type presumptions based on the file extension. The content type can describe anything; application XML, user XML (see discussion in section 3.1), images, sounds, video, or any other binary objects. Every part must be connected to some other part using a relationship. Inside package are special XML files with a ".rels" extension which defines relationship between parts. There is also a *start part* (sometimes called "root", although the graph containing all parts isn't necessarily a tree structure). Fig. 2 depicts the structure of a package.[4]

[3] http://www.ecma-international.org/publications/standards/Ecma-376.htm

[4] http://www.codeproject.com/KB/office/OpenXML.aspx

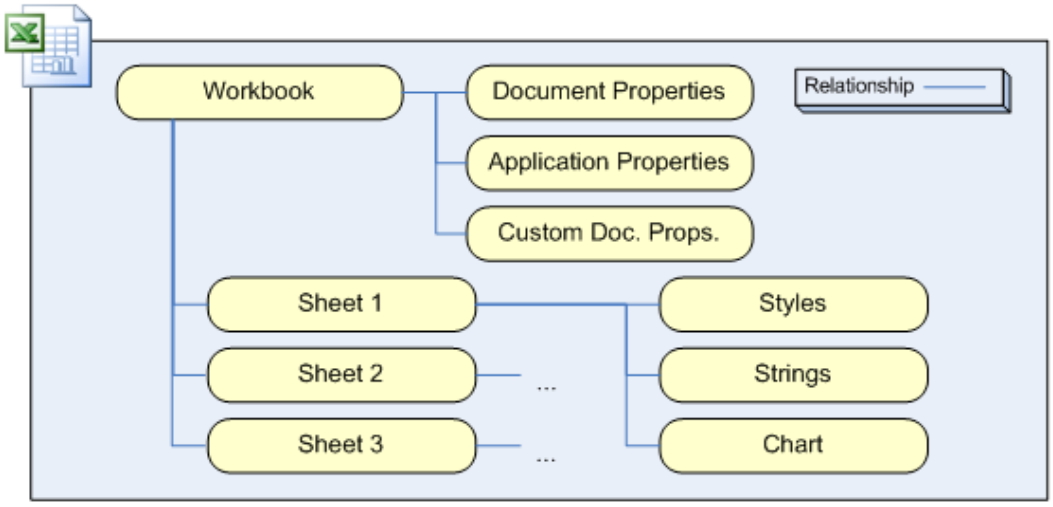


Figure 2. The structure of an Office Open XML package

### 3.1 Microsoft "Custom XML" Tag in Office Open

Custom XML markup *is about embedding custom XML defined outside of Office Open XML to support solution which aim to structure a document using business semantics, not only using formatting.*[5] For example, suppose we want to annotate a certain element using a *CustomerName* element, so that a separate tool can easily locate the customer name information afterwards. Because Office Open XML files conform to XML rules, a custom element would violate the schema, not being mentioned there. Hence enters the standard *CustomXML* element, with an attribute that points to the real custom XML element.

For example, the following listing adds a custom element named *lorem.*

```
<w:customXml w:element="lorem">
 <w:r>
  <w:t xml:space="preserve">Lorem ipsum dolor ... pharetra eget, diam.</w:t>
 </w:r>
 </w:customXml>
```

Conceptually, custom tags could provide a jumping point to business logic from within the document. In practice however, this is not a satisfactory solution of the ultimate MVC architectural goal, for the following reasons:

1. The *customXML* element is only supported by .docx, not by .pptx or .xlsx files.

2. Even for those .docx documents that benefit from a *customXML* element, the document cannot share the controller (business logic) with other documents.

[5] http://www.zdnet.com/blog/microsoft/custom-xml-the-key-to-patent-suit-over-microsoft-word/3712

# 4. Implementation Issues

## 4.1 Implementation Objectives

1. Cater for a *thin client* web-based capability. A naive implementation towards this goal would be a web based MVC solution using a workflow similar to the one depicted in Fig. 1 (probably using raw XML model files instead of a DB).

2. Backward compatibility using techniques such as virtualization and "thin-app" delivery of existing applications. A thin-app solution has a preconfigured image of an application, such as MS Word 2010, available on the cloud or intranet. It is transferred to the client whenever they choose to open a .docx file[6].

3. Enjoy the improved robustness, flexibility, and coordination features promised by an MVC architecture, as discussed in section 2.

## 4.2 Envisioned MVC Relationships

We envision the view-controller and controller-model relationships depicted in the class diagram of Fig. 3. Accordingly:

a. The controller is capable of picking up one of many possible rendering views, based on its business logic and user inputs; this is the prevailing approach with modern software development environments such as Eclipse.

b. A controller can be shared by many model files, such as all NPS PO's sharing the same tax rate calculation controller.


Figure 3. Envisioned MVC relationships.

Clearly, to be able to calculate complex business logic, the controller must be able to access an executing application, process, or thread. Note that the controller application is distinct from the *document application*; the later is akin to the contemporary MS Word, MS Excel, or Adobe Acrobat which obviously do not work on MVC storage documents. In contrast, the controller application can be developed by the user's organization and is independent of the document application.

Note that both the model and the controller can have relationships with other entities:

- The model can extend and have relationships to other model files[7]

[6] Note that virtualization is mostly concerned with the location of the *application* and *machine*, not the data, which can reside on the client.

[7] D. Drusinsky, J.B. Michael, T.W. Otani and M. Shing, Putting Order Into the Cloud: Object-oriented UML-based Rule Enforcement for Document and Application Organization, NPS-CS-10-009.

- The controller can use services provided by other executables.

### 4.3 Serialization

Serialization is the process of converting a data structure or object into a sequence of bits to be stored in a file. We envision two primary approaches to serialization:

a. Storing the model, view, and controller as separate entities on the cloud. The model and view are envisioned as being but raw XML files. The controller is envisioned to be serialized as an XML file (called the *controller file*) with:

   - A specially marked element that points to the corresponding controller application or executable. This is similar to the way JSP points to the class *com.devsphere.examples.mapping.simple.SimpleBean* executing JSP calls in listing 1 below.
   - Slots to be populated by the controller, such as tax rate. This is similar to the way web applications work with JSP and ASP, having marks html so called *slots* to be populated by a server side computation, which are shown in bold in listing 1.

```
<%@ page language="java"%>
<jsp:useBean id="simpleBean" scope="request"
    class="com.devsphere.examples.mapping.simple.SimpleBean"/>
<HTML>
<HEAD><TITLE>Simple bean</TITLE></HEAD>
<BODY>
<H3>Simple Example</H3>
<P><B> SimpleBean properties: </B>
    <P> string = <jsp:getProperty name="simpleBean"
         property="string"/>
    <P> number = <jsp:getProperty name="simpleBean"
         property="number"/>
    <P> integer = <jsp:getProperty name="simpleBean"
         property="integer"/>
    <P> flag = <jsp:getProperty name="simpleBean"
         property="flag"/>
    <P> colors = <%= toString(simpleBean.getColors()) %>
    <P> list = <%= toString(simpleBean.getList()) %>
    <P> optional = <jsp:getProperty name="simpleBean"
         property="optional"/>
</BODY>
</HTML>
```

Listing 1. JSP populating slots within html code.

   - URI's to the corresponding model and view files as prescribed by the relationships of Fig. 3.

   For lack of a better name, we call this option the *MVC storage* option.

b. The *monolithic file* approach whereby the model, view, and controller are stored as a single file, as done with contemporary .doc, .docx, .pdf documents and others.

Clearly, the MVC storage approach is superior because the monolithic file approach suffers from the disadvantages specified in section 3.1.

With the MVC storage approach however, support for backward compatibility (i.e. the objectives discussed in section 4.1) requires an ability to compose a monolithic file from an MVC storage representation and vice-versa.

### 4.4 Thin Client Implementation Approach

This implementation approach is but a mirror image of the current MVC architecture for Web based applications depicted in Fig. 1, with the following exceptions:

1. The protocol isn't necessarily http.

2. The canvas isn't necessarily a web browser, although using the browser as the canvas works well for other objectives of the DoD cloud effort.

3. The model is stored as a raw XML file.

### 4.5 Short-term, Backward Compatible Implementation

This implementation approach composes a conventional document file (e.g., a .docx file) from the MVC storage prior to the invocation of the application (e.g., MS Word), and converts it to an MVC storage representation on the back end, whenever the application performs a save operation. A proof of concept of this workflow is underway.

### 4.6 Thin-data

The term thin-data was coined by the author to resemble the thin-app term described earlier. With a thin-app, the application is stored on the cloud, loaded as an image to the desktop on demand , and executed on the desktop, using the desktop's file system; when the application terminates the application's image is cleared from the desktop (while data persists).

Similarly, a thin-data application stores its data on the cloud, loads it on the desktop on demand, and saves the data back to the cloud when the application terminates. Whether the data clears from the desktop after the application terminates is probably not a hardwired property, but parameterized, depending on security concerns and connectivity, as discussed in Section 4.7.

To demonstrate thin-data, consider Google Docs. In its current configuration, an end user can open an MS Word document that is stored on Google Docs using MS Word. Missing towards the thin-data goal however, is the capability to automatically upload the saved Word document back to the cloud whenever the user saves the document or when Word terminates.

As the Google Docs Example shows, thin-app and thin-data are orthogonal capabilities. Clearly, as discussed in section 4.5, thin-data is well suited for backward compatibility.

Another important feature of thin-data is that it can work hand in hand with the thin client solution in that a user can seamlessly interleave and interchange the use of thin-client and a backward compatible desktop application (or thin-app; e.g. MS Word) on the

same data, provided that when the user uses the desktop application they use the thin-data approach so the data is serialized on the cloud.

### 4.7 Flexibility and Lack of Connectivity

The integration of thin client, thin-app, and thin-data provides some flexibility that can prove useful when the system is stressed. For example, consider the situation where a certain organization is disconnected from the internet. Suppose the organization has a small pool of licenses for Office applications that are deployable via thin-app from a local server, without needing full connectivity to the cloud; we call these instances *emergency instances*. Recall now that thin-app has an option in which the local instance of the data is not erased after the application terminates; this instance is like an *emergency version of the data*. Hence, when the user uses emergency thin-app instances of the application together with the emergency version of the data, they can operate while connectivity is unavailable. When connectivity is resumed, the end user resumes using the thin-client application; it senses that the emergency version of the data is more recent than the MVC storage version, and therefore makes the appropriate decision as to which version of the data to use. When the thin-client terminates, the MVC storage is updated and becomes the most recent.

An alternative implementation of emergency license instances is ticket-based instances, which operate a limited number of times, and can only be used with a supervisor's authorization.

THIS PAGE INTENTIONALLY LEFT BLANK

# INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Research Sponsored Programs Office, Code 41
   Naval Postgraduate School
   Monterey, CA 93943

4. Professor Peter Denning
   Naval Postgraduate School
   Monterey, California

5. Professor Doron Drusinsky
   Naval Postgraduate School
   Monterey, California

6. Professor Bret Michael
   Naval Postgraduate School
   Monterey, California

7. Professor Thomas Otani
   Naval Postgraduate School
   Monterey, California

8. Professor Man-Tak Shing
   Naval Postgraduate School
   Monterey, California

9. Mr. John Shea
   Office of the DoD CIO
   Arlington, Virginia

10. COL Kevin Foster, USA
    Office of the DoD CIO
    Arlington, Virginia

11. Professor George Dinolt
    Naval Postgraduate School
    Monterey, California

12. Professor Loren Peitso
Naval Postgraduate School
Monterey, California

13. Mr. Alex Nelson
Naval Postgraduate School
Monterey, California

14. Mr. Scott J Dowell
Computer Science Corporation
San Diego, California

15. Mr. Michael Lee
Touchstone Consulting Group
Washington, D.C.

16. Ms. Karen Gordon
Institute for Defense Analyses
Alexandria, Virginia

17. Dr. Jeffrey Voas
National Institute of Standards and Technology
Gaithersburg, Maryland

18. Dr. Mark Lee Badger
National Institute of Standards and Technology
Gaithersburg, Maryland

19. Dr. Tim Grance
National Institute of Standards and Technology
Gaithersburg, Maryland